# Plus courts chemins dans un graphe

*Vendredi 3 février 2017*

## Buts du TP

— (Se remettre en tête la méthode d'Euler).
— Lire des données dans un fichier.
— Mettre en place trois algorithmes pré-machés sur les graphes.
— Et pourquoi pas, essayer de les comprendre (avant le TP) !

**Exercice 1.** *Créer (au bon endroit) un dossier associé à ce TP. Lancer Spyder/Pyzo/Idle, sauvegarder au bon endroit le fichier* `tdprobasouuntruccommeça.py`*; écrire une commande absurde, de type* `print( 6*7 )` *dans l'éditeur, sauvegarder et exécuter.* Si vous êtes sous Spyder, changez (via F6) les options d'exécution, pour avoir à chaque exécution une nouvelle console et garder la main dessus. (il y a donc deux cases à vérifier/cocher). *Dans le dossier du TP du jour, placer les fichiers suivants :*

`cadeau-graphes.py` `g5.txt` `g100.txt`

**Exercice 2.** *Vérifier que le premier exo a effectivement bien été fait : tout manquement donnera lieu à une agitation néfaste pour tout le monde...*

## 1 Méthode d'Euler

On rappelle que la méthode d'Euler (explicite) consiste à approcher la solution du « problème de Cauchy » $\begin{cases} y'(t) = f(t, y(t)) \\ y(a) = \alpha \end{cases}$ sur un intervalle $[a, b]$ en prenant un pas de temps $h = \dfrac{b-a}{n}$ (avec $n > 0$ à discuter), et en construisant les valeurs $y_0, ..., y_n$ censées approcher $y(a + kh)$ (avec $0 \leqslant k \leqslant n$) en prenant $y_0 = \alpha$, puis :

$$\forall k \in [\![0, n-1]\!] \qquad y_{k+1} = y_k + hf(t_k, y_k), \quad \text{où} \quad t_k = a + kh = a + k\frac{b-a}{n}\cdot$$

**Exercice 3.** *En reprenant et complétant le code proposé dans le fichier* `cadeau-graphes.py`*, écrire une fonction* `euler`*, telle que l'appel* `euler(f, a, b, n, y0)` *retourne la liste des $y_k$ décrite plus haut.*
*Tester ce code pour le problème de Cauchy* $\begin{cases} y'(t) = 1 + y(t)^2 \\ y(0) = 0 \end{cases}$ *(pour lequel on connaît la solution : c'est la fonction tangente) sur* $[0, 3/2]$.

Le code fourni dans `cadeau-graphes.py` permettra normalement d'obtenir le graphique suivant :

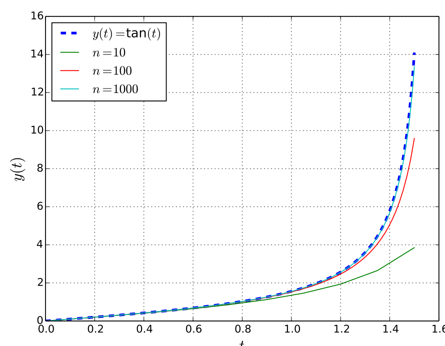


FIGURE 1 – Méthode d'Euler pour $y'(t) = 1 + y(t)^2$

## 2 Plus courts chemins dans un graphe

On va dans cette partie déterminer les distances mutuelles entre les différents sommets du graphe suivant (noté $G_5$ dans la suite) ainsi que pour un autre graphe à 100 sommets :

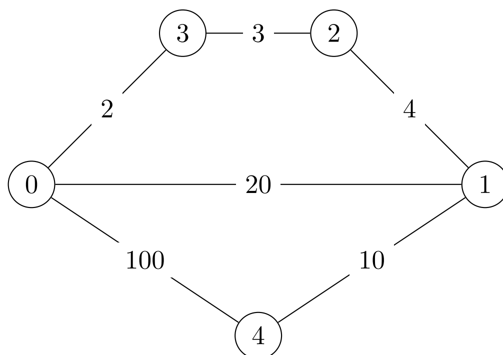


Figure 2 – Le graphe $G_5$

La distance entre deux sommets est la somme minimale des poids des arêtes sur un chemin reliant ces deux sommets. Par exemple, le chemin optimal entre les sommets 0 et 1 a pour somme de poids 9 (et on ne peut pas faire mieux), donc $d(0,1) = 9$. On dit que ce chemin optimal est de longueur 3 (alors qu'on dispose d'un chemin non optimal de longueur 1 mais de poids 20).
*Oui, bon, c'est un peu pénible, mais la longueur va désigner le nombres d'arêtes, et la distance la somme des poids des arêtes.*

**Exercice 4.** *Calculer à la main les différentes distances (« on voit bien que... ») entre les couples de sommets, et les indiquer dans cette matrice qu'on complétera ($D_{i,j}$ contient $d(i,j)$ en indexant lignes et colonnes depuis 0) :*

$$D = \begin{pmatrix} 0 & 9 & & & \\ & 0 & & 7 & \\ & & 0 & & \\ & & & 0 & \\ & & & & 0 \end{pmatrix}$$

On pourra se contenter de remplir la partie supérieure, du fait de la symétrie.

Petits rappels techniques pour la suite :
- On peut facilement créer une matrice $(n,n)$ constituée de 0 via `[[0] * n for _ in range(n)]`
- On peut dupliquer une matrice (liste de liste) existante via la fonction `deepcopy` de la bibliothèque `copy`

### 2.1 Récupération de données

Les graphes sont fournis dans des fichiers sous le format suivant : une première ligne donne le nombre $n$ de sommets, qu'on indexe de 0 à $n-1$, puis les $n$ lignes suivantes fournissent les voisins (avec les poids) pour chaque sommet. Par exemple, le graphe $G_5$ vu plus haut est codé de la façon suivante dans un fichier[1] :

```
--------------------
5
-,20,-,2,100
20,-,4,-,10
-,4,-,3,-
2,-,3,-,-
100,10,-,-,-
--------------------
```

1. Sans les tirets !

Dans la deuxième ligne, on voit donc qu'il y a 3 arêtes issues de 0, arrivant aux sommets 1 (poids 20), 3 (poids 2) et 4 (poids 100).
La première étape va consister à mettre ces données dans une matrice. Pour signifier qu'il n'y a pas d'arête entre $i$ et $j$, un bon point de vue consisterait à créer un « entier » spécial qu'on noterait $\infty$. MAIS pour simplifier les choses, on va décider de placer $10^8$ pour toute arête inexistante (un espèce de mur, donc).
On va écrire une fonction `lecture_fichier` qui fera ce travail :

```
>>> g5 = lecture_fichier('g5.txt')
>>> g5
[[100000000, 20, 100000000, 2, 100], [20, 100000000, 4, 100000000, 10],
[100000000, 4, 100000000, 3, 100000000], [2, 100000000, 3, 100000000, 100000000],
[100, 10, 100000000, 100000000, 100000000]]
```

On rappelle le principe de lecture dans un fichier texte :

— On ouvre le fichier en lecture via `f = open( le_nom, 'r')`

— La ligne courante peut être lue via `ligne = f.readline()` (et la nouvelle ligne courante... est la suivante !)

— *On peut lire toutes les lignes restantes à partir de la ligne courante via une boucle du type :*

```
for ligne in f.readlines():
    bli
    bla
```

*Mais ici, on pourra se passer d'une telle construction, le nombre de lignes à lire étant vite connu.*

— On ferme le fichier courant via `f.close()`

Ensuite, pour analyser une ligne (qui est une chaîne de caractères) :

— On peut nettoyer les éventuelles saletés à la fin (typiquement : un passage à la ligne `\n`, qui termine systématiquement les lignes d'un fichier texte) via la méthode `strip` : `ligne_propre = ligne.strip()`

— On peut ensuite la casser selon un séparateur via la méthode `split` :

```
>>> "bli;bla;blu".split(';')
['bli', 'bla', 'blu']
```

— Pour transformer une chaîne en entier, rien de plus simple :

```
>>> int( ' 1515 ')
1515
```

À vous de jouer !

**Exercice 5.** *Écrire une fonction* `lecture_fichier` *prenant en entrée un nom de fichier, ouvrant ce fichier et calculant à partir du contenu la matrice d'adjacence du graphe codé dans ce fichier.*
Le résultat retourné sera une liste de listes, comme dans l'exemple de $G_5$ vu plus haut.

On note dès maintenant que face à un graphe représenté ainsi, on obtient facilement le nombre de sommets en demandant la longueur de la liste.

```
>>> g5 = lecture_fichier('g5.txt')
>>> len(g5)
5
```

Dans toute la suite de la présentation, $n$ désigne toujours le nombre de sommets (5 sur l'exemple), et $G$ représente la matrice du graphe d'adjacence. Au niveau de Python, ce qu'on appelle *matrice* désignera toujours une liste de listes, les indexations commençant donc à 0.

## 2.2 Méthode élémentaire

On va calculer de proche en proche (pour $k$ allant de 1 à $n-1$), les matrice $M^{(k)}$ représentant en position $(i,j)$ la distance entre les sommets $i$ et $j$ **via des chemins de longueur au plus** $k$ :

— Déjà, $M^{(1)}_{i,j} = \begin{cases} 0 & \text{si } i = j \\ G_{i,j} & \text{sinon} \end{cases}$

— Ensuite :

$$\forall k \in [\![2, n-1]\!]\ \forall i,j \in [\![0, n-1]\!], \qquad M^{(k)}_{i,j} = \min\left(M^{(k-1)}_{i,j}, \min\left\{M^{(k-1)}_{i,\ell} + G_{\ell,j} \;\middle|\; 0 \leqslant \ell \leqslant n-1\right\}\right)$$

**Exercice 6.** *Expliquer ces formules !*

**Exercice 7.** *Expliquer comment on peut ainsi calculer toutes les distances mutuelles entre sommets en $O(n^4)$ opérations élémentaires.*

**Exercice 8.** *Programmer effectivement cet algorithme.*
*On commencera par écrire une fonction* `operation` *telle que pour deux matrices $A$ et $B$, l'appel* `operation(A, B)` *renvoie la matrice $C$ telle que $C_{i,j} = \min\left(A_{i,j}, \min(A_{i,\ell} + B_{\ell,j},\ 0 \leqslant \ell \leqslant n-1)\right)$.*

Voici ce que vous devez obtenir comme matrice de distances :

```
>>> distances1(g5)
[[0, 9, 5, 2, 19], [9, 0, 4, 7, 10], [5, 4, 0, 3, 14], [2, 7, 3, 0, 17], [19, 10, 14, 17, 0]]
```

## 2.3 Floyd-Warshall

L'algorithme de Floyd-Warshall [2] est une petite variante de l'algorithme naïf : on calcule cette fois les $N^{(k)}$ représentant en position $(i,j)$ la distance entre les sommets $i$ et $j$ **via des sommets intermédiaires strictement majorés par** $k$ :

— Déjà, $N^{(0)}_{i,j} = \begin{cases} 0 & \text{si } i = j \\ G_{i,j} & \text{sinon} \end{cases}$

— Ensuite :

$$\forall k \in [\![1,n]\!]\ \forall i,j \in [\![0,n-1]\!], \qquad N^{(k)}_{i,j} = \min\left(N^{(k-1)}_{i,j}, N^{(k-1)}_{i,k-1} + N^{(k-1)}_{k-1,j}\right)$$

**Exercice 9.** *Expliquer les formules précédentes.*

**Exercice 10.** *Expliquer comment on peut ainsi calculer toutes les distances mutuelles entre sommets en $O(n^3)$ opérations élémentaires.*

**Exercice 11.** *Programmer effectivement cet algorithme.*
*On commencera par écrire une fonction* `operationFW` *telle que pour une matrice $A$ et un entier $k \in [\![1,n]\!]$, l'appel* `operationFW(A, k)` *renvoie la matrice $B$ telle que $B_{i,j} = \min\left(A_{i,j}, A_{i,k-1} + A_{k-1,j}\right)$.*

```
>>> distances2(g5)
[[0, 9, 5, 2, 19], [9, 0, 4, 7, 10], [5, 4, 0, 3, 14], [2, 7, 3, 0, 17], [19, 10, 14, 17, 0]]
```

## 2.4 Dijkstra

Dans l'algorithme de Dijkstra (1956), on calcule la distance d'un sommet à tous les autres en $O\left(|A| \ln |S|\right)$ opérations élémentaires, avec les structures de données adaptées (liste d'adjacence et structure de tas – très largement hors programme). Avec nos braves matrices d'adjacences et recherches linéaires de minimum, on va obtenir un algorithme en $O(|S|^2) = O(n^2)$.
Le principe est de réaliser une partition des sommets $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$, avec $\mathcal{S}_1$ qui grossit (il est initialisé à $\{s_0\}$) en préservant l'invariant suivant :

— la distance des $s \in \mathcal{S}_1$ à $s_0$ est connue ;
— pour $s \in \mathcal{S}_2$, on connaît seulement la distance à $s_0$ via $\mathcal{S}_1$ (tous les sommets intermédiaires étant dans $\mathcal{S}_1$).

On va réaliser $n-2$ fois les opérations suivantes :

— on choisit $s_1$ le (un) sommet de $\mathcal{S}_2$ à plus petite distance de $s_0$ ;
— on le bascule dans $\mathcal{S}_1$ ;
— pour chacune des arêtes $(s_1, s_2)$ avec $s_2 \in \mathcal{S}_2$, on met à jour la nouvelle distance de $s_2$ à $s_0$ via $\mathcal{S}_1$ :

$$d(s_2) \leftarrow \min\left(d(s_2), d(s_1) + w\right)$$

(où $w$ est le poids de l'arête $(s_1, s_2)$).

Après ces $n-2$ étapes, $|\mathcal{S}_1| = n-1$, et les distances à $s_0$ via $\mathcal{S}_1$ sont donc exactement les distances à $s_0$.

2. 1959 – l'auteur réel étant un certain Bernard Roy !

**Exercice 12.** *Exécuter cet algorithme à la main pour* $s_0 = 0$ *dans le graphe* $G_5$.

**Exercice 13.** *Écrire une fonction* `sommet_mini` *prenant en entrée la liste des distances (courante) à* $s_0$ *via* $\mathcal{S}_1$ *et la liste des sommets de* $\mathcal{S}_2$, *et qui retourne la valeur du (d'un) sommet de* $\mathcal{S}_2$ *de plus petite distance à* $s_0$ *via* $\mathcal{S}_1$.
*Don't panic : c'est en fait très simple !*

On y est presque !
Arrivé ici, on peut noter qu'il est inutile de tenir à jour la liste $\mathcal{S}_1$ : seule la liste $\mathcal{S}_2$ nous intéresse.

**Exercice 14.** *Écrire une fonction* `Dijkstra` *prenant en entrée un graphe, un sommet, et calculant les distances de ce sommet à tous les autres dans le graphe via l'algorithme de Dijkstra.*

**Exercice 15.** *Faire des tests, en comparant les différentes valeurs et temps d'exécution sur* $G_5$ *et sur* $G_{100}$ *pour les trois algorithmes.*

```
>>> Dijkstra(g100, 0) == distances2(g100)[0]
True
```